

Temporal and Modal Logic Programming Languages*

Manolis Gergatsoulis

Institute of Informatics & Telecommunications,
National Centre for Scientific Research (N.C.S.R.) ‘Demokritos’,
153 10 Aghia Paraskevi Attikis, Greece
e_mail: manolis@iit.demokritos.gr

Abstract

Temporal and modal logics have been used in many applications in Artificial Intelligence and Computer Science for the manipulation of information with time-dependent or, in general, context-dependent properties. Knowledge representation and reasoning, temporal planning, simulation, temporal verification, and description of agent systems, are among the applications for which temporal and modal logics have been proven useful. Programming languages based on Temporal or Modal Logics, provide powerful executable formalisms for implementing such applications. In this article we introduce the basic notions behind temporal and modal logic programming languages. We briefly present representative temporal and modal logic programming languages and give examples of their use.

Keywords: Temporal logic programming, modal logic programming, programming languages.

1 Introduction

The main idea behind *logic programming* is the use of (a subset of) logic as a programming language. In the well-known logic programming language Prolog, the knowledge and assumptions about a problem are formalized as a set of logical axioms in the form of *Horn clauses* [Llo87]. A proof system, called *SLD-resolution*, is used to prove that a statement, called a *goal clause*, is a logical consequence of the program. In this way, SLD-resolution plays the role of an execution mechanism of the language.

*This article appears in A. Kent and J. G. Williams (editors) ‘‘Encyclopedia of Microcomputers’’, Volume 27, Supplement 6, pages 393-408, Marcel Dekker, Inc., New York., 2001.

The success of *logic programming* has motivated several extensions, with some of them forming a programming paradigm on its own. Among other well-known extensions (such as *constraint logic programming* [JM94], *disjunctive logic programming* [LMR92], *higher order logic programming* [NM94], etc.), there are extensions toward formalisms allowing explicit reasoning about time, space, and, in general, context-dependent information. This family of logic programming languages forms the research areas of *temporal* and *modal logic programming*.

Temporal and modal logic programming languages are based on *temporal logics* [Gol92, GHR94] and *modal logics* [Che80], respectively. In analogy to classical logic programming, syntactic subsets of temporal and modal logics which present well-defined computational behavior are used as the basis for developing programming languages.

Declarative semantics, which are extensions of van Emden & Kowalski semantics [vEK76], are usually proposed for temporal and modal logic programming languages. The operational semantics of most temporal and modal logic programming languages are based on theorem proving techniques developed for the underlying temporal and modal logics. However, the operational semantics of some of the programming languages proposed in the literature are based on the idea of translating the programs into classical logic programs or constraint logic programs and then using classical theorem proving techniques.

In this article, we introduce the basic ideas underlying temporal and modal logic programming languages and present some representative languages of this family. Section 2 presents temporal and modal logics and their properties. Section 3 presents representative temporal logic programming languages, together with simple examples of their use. Section 4 presents modal logic programming languages. Section 5 gives references to other papers referring to modal and temporal logic programming languages. Finally, section 6 concludes the article.

2 Modal and Temporal Logics

Modal logics [Che80] are logics of qualified truth. The language of a modal logic includes all classical logical symbols as well as some *modal operators* such as \Box (*necessary*) and \Diamond (*possible*). Modal formulas include all classical formulas and the formulas obtained by formation rules of the form

If A is a formula and ∇ is a modal operator, then ∇A is a formula.

In a modal logic, the meaning of a formula depends on an implicit *context*. The collection of contexts is said to be the set of *possible worlds* denoted by \mathcal{U} . In Kripke semantics, a relation over \mathcal{U} , called the *accessibility relation*, is associated with each modal operator. A world v is said to be *accessible* from a world u if $\langle u, v \rangle$ belongs to

the accessibility relation. A (modal) *interpretation* of a modal language assigns meaning to all elements of the language. In order to define a modal interpretation, we need a nonempty set D , called the *domain* of the interpretation. We assign to each constant in the language an element of D , to each function symbol a function from D^n to D , and to each predicate symbol a function from \mathcal{U} to 2^{D^n} . In order to assign a truth value to a formula, we combine the truth values of its subformulas. For subformulas composed using classical symbols such as \neg , \wedge , \vee , \rightarrow , \leftrightarrow , as well as the quantifiers \forall and \exists , the truth values of the subformulas are combined in the usual way (as in classical logic) to give the truth value of the formula. For formulas obtained by applying modal operators to other formulas (subformulas), we have to define the meaning of modal operators. In Kripke semantics, the meaning of modal operators is determined by their accessibility relations. Consider, for example, the modal operator \Box (necessary). Then, $\Box A$ is true at a world w if the formula A is true at all worlds that are accessible from w . Similarly, for the modal operator \Diamond (possibly), we have that $\Diamond A$ is true at a world w if the formula A is true at some world which is accessible from w .

The elements of the language whose values do not depend on the context are said to be *rigid*, whereas the symbols whose values depend on the context are said to be *flexible*. Usually, in the modal and temporal logics on which modal and temporal logic programming languages are based, constants and function symbols are considered as rigid, whereas predicate symbols are flexible.

Temporal logic [Gol92] can be considered as an instance of modal logic in which the set of possible worlds models the collection of moments in time. The accessibility relations assigned to temporal operators in a temporal logic exhibit the properties of time. The most important of these properties relate to the following questions:

Time points or time intervals? *Time points* are considered as primitives in some temporal reasoning systems, whereas in other systems, *time intervals* are considered as primitives. In systems which consider time points as primitives, intervals are defined as pairs of time points, corresponding to the lower and the upper ends of the interval.

Bounded or unbounded time? Time can be bounded or unbounded. We say that time is *unbounded* to the future or to the past if every time is succeeded by a later or earlier time, respectively. Otherwise, if there is a last or first time, we say that time is *bounded* in the future or in the past, respectively.

Discrete, dense, or continuous time? The time domains chosen can map times either to the set of integers \mathcal{Z} or to the set of rational numbers \mathcal{Q} or to the set of real numbers \mathcal{R} and the time is said to be *discrete*, *dense*, or *continuous*, respectively.

Linear or branching time? When time points are considered, time is *linear* if the set of time points is totally ordered. If the set of time points is partially ordered, then time is said to be *branching*. The idea of branching time has been put forward as a way of handling uncertainty about the future. In branching time from each moment, there exist

many possible alternative futures.

3 Temporal Logic Programming

3.1 Linear-Time Temporal Logic Programming

The logic programming languages presented in this section are based on linear-time temporal logics.

3.1.1 Chronolog

Chronolog [Org91, OW92a, OW92b, OW93, OWD93] is a temporal logic programming language based on a simple temporal logic in which time is linear and discrete. Chronolog has two temporal operators: The operator **first**, which refers to the first moment in time, and the operator **next**, which refers to the next moment in time. Chronolog uses the set \mathcal{N} of natural numbers as the collection of moments in time. A Chronolog program is a set of *temporal clauses* of the form

$$A \leftarrow B_1, B_2, \dots, B_n$$

where A, B_1, \dots, B_n are temporal atoms. A *temporal atom* is an atomic formula preceded by a number of temporal operators.

As an example, consider the following Chronolog program simulating the operation of the traffic lights:

```
first light(green).
next light(amber) ← light(green).
next light(red) ← light(amber).
next light(green) ← light(red).
```

The declarative semantics of Chronolog has been developed using the notion of *temporal Herbrand interpretations*. A temporal Herbrand interpretation of a program P is a subset of its *temporal Herbrand base* B_P , where B_P is the set of all ground canonical atoms which can be constructed using predicates appearing in P and ground terms in the Herbrand universe U_P of P as arguments. The Herbrand universe U_P of a Chronolog program P consists of all ground terms which can be constructed using constants and function symbols from P . A temporal atom is said to be *canonical* if it is of the form **first next^t A**, for some $t \geq 0$, where A is a classical atom and **next^t** represents t successive applications of **next**. It has been shown [Org91, OW92a] that every Chronolog program has a unique *minimal temporal Herbrand model*. This model contains all ground canonical temporal atoms which are logical consequences of the program. A fixpoint characterization of the minimal temporal Herbrand model is also given in [Org91, OW92a].

The execution mechanism of Chronolog is a temporal version of SLD-resolution called *TiSLD-resolution* [OW93]. TiSLD-resolution is applied to canonical program and goal clauses. Canonical program clauses are *canonical instances* of the program clauses obtained by applying a sequence of temporal operators of the form `first nextt`, for some $t \geq 0$, to all atoms of a program clause. TiSLD-resolution is applied to canonical program and goal clauses of a Chronolog program in the same way as SLD-resolution is applied to program and goal clauses of a classical Horn clause program. Consider, for example, the canonical goal clause G_0 :

$$\leftarrow \text{first next light(L)}.$$

asking for the color of the light at the second moment in time. The temporal atom in G_0 can be matched with the head of the canonical instance:

$$\text{first next light(amber)} \leftarrow \text{first light(green)}.$$

of the second clause in the traffic light program. The most general unifier in this matching is $\theta_0 = \{L/\text{amber}\}$ and the new goal obtained is G_1 :

$$\leftarrow \text{first light(green)}.$$

Now, the temporal atom in G_1 can be matched with the first clause in the program, which is in canonical form, giving, in this way, the empty goal. The substitution in this matching is the empty substitution $\theta_1 = \{\}$. The answer computed in this derivation is the composition of the substitutions θ_0 and θ_1 which gives $L = \text{amber}$.

TiSLD-resolution is a sound and complete proof procedure.

Extensions of Chronolog have been introduced in [OWD93, OW94]. In particular, in [OWD93] Chronolog(\mathcal{Z}), is introduced, which is based on a linear-time temporal logic with unbounded past and future with the set \mathcal{Z} of integers as the collection of moments in time. Besides the temporal operators `first` and `next` of Chronolog, Chronolog(\mathcal{Z}) has also an operator, denoted by `prev`, to look into the past. In the extension of Chronolog proposed in [OW94], to each predicate p appearing in the program, a *choice predicate* $\#p$ is associated representing a single valued version of p . The value selected by a choice predicate is arbitrarily chosen among the values returned by the corresponding conventional predicate.

3.1.2 Disjunctive Chronolog

Disjunctive Chronolog [GRP96] is an extension of Chronolog in which the heads of the clauses may be disjunctions of temporal atoms rather than single temporal atoms. Disjunctive Chronolog combines the ideas of both Chronolog and Disjunctive Logic Program-

ming [LMR92]. Using Disjunctive Chronolog, we can express uncertainty as shown in the following program:

```

first visit(john,greece) ∨ first next visit(john,greece).
have_good_time(X) ← visit(X,greece).

```

The uncertainty is expressed by the first clause which says that “*John is either going to visit Greece this or next year (or both)*”. We call this form of uncertainty *time uncertainty* in contrast to another form of uncertainty, which we call *event uncertainty*, in which we have a specific time point in which an event occurs but we do not know exactly which one of a number of possible events has taken place.

The semantics of Disjunctive Chronolog extend the semantics of both Chronolog [Org91] and Disjunctive Logic programming [LR91, LMR92]. More specifically, minimal model semantics and fixpoint semantics have been defined for Disjunctive Chronolog programs [GRP96, GRP97b] and their equivalence has been shown. Proof procedures for Disjunctive Chronolog have been also investigated in [GRP97b], where it is shown that proof procedures developed for classical disjunctive logic programs [MRL91, LR91, LMR92], can be extended to apply to Disjunctive Chronolog programs.

3.1.3 Templog

Templog was proposed by Abadi & Manna [AM89]. Templog is based on a discrete linear time with unbounded future and uses the set \mathcal{N} of natural numbers as the collection of moments in time.

Templog extends classical Horn logic programming to allow the use of the temporal operators \bigcirc (next), \square (always), and \diamond (eventually). *Templog programs* are sets of *Templog clauses*. Templog clauses extend classical Horn clauses by allowing *next-atoms* (i.e., atoms preceded by any number of \bigcirc 's) to appear everywhere atoms appear in Horn clauses and by allowing the use of \diamond in the body of the clauses and of \square in the head of clauses or in front of entire clauses.

Consider, for example, the following Templog program, which defines the Fibonacci numbers:

```

fib(0).
 $\bigcirc$  fib(1).
 $\square(\bigcirc \bigcirc \text{fib}(Z) \leftarrow \text{fib}(X), \bigcirc \text{fib}(Y), Z \text{ is } X + Y)$ .

```

Declarative semantics have been developed for Templog programs which are a temporal extension of the van Emden & Kowalski semantics [vEK76]. The execution mechanism of Templog is a refutation procedure, similar to SLD-resolution for classical logic programming [Llo87], called *TSLD-resolution*. Every step in TSLD-resolution consists of resolving a *candidate* next-atom from the current goal with the head of a program clause, to produce a new goal. A goal is supposed to be in its *canonical form*. In the canonical

form of a goal (or clause body), each occurrence of \diamond always has in its scope at least one next-atom that is in the scope of no other \diamond . A next-atom occurring in a goal is said to be a *candidate* next-atom if it is in the scope of at most one \diamond in the canonical form of the goal. It can be proved that there is at least one candidate next-atom in any nonempty goal. TSLD-resolution is a sound and complete proof procedure for Templog [Bau93].

Alternatively, as Brzoska showed in [Brz91], Templog programs can be considered as CLP(\mathcal{A})-programs over a suitable algebra \mathcal{A} . A meaning-preserving transformation for the translation of Templog programs and goal clauses into classical constraint logic programs and goal clauses, respectively, is also given in [Brz91].

3.1.4 Metric Temporal Logic Programming

Brzoska [Brz98, Brz93] investigated temporal logic programming languages based on *metric temporal logics*. The time in Brzoska's work is linear and may be either discrete or dense. A fragment of metric temporal logic which can be handled within the constraint logic programming framework is the so-called *Simple Metric Temporal Logic programs* (*MTL-programs* for short). The syntax of MTL-programs in BNF notation is as follows:

The goal clauses, called *simple MTL-goals*, are defined by

$$G ::= \epsilon \mid A \mid \diamond_I A \mid G \wedge G$$

whereas the Horn formulas, called *simple MTL-Horn formulas*, are defined by

$$D ::= A \mid \square_I D \mid D \leftarrow G$$

where I denotes an interval, ϵ represents the empty goal and A ranges over atoms. An interval I is of the form $[c^-, c^+]$ with $c^-, c^+ \in \mathcal{Z} \cup \{-\infty, +\infty\}$ in the discrete case, and $I \in \{[c^-, c^+], (c^-, c^+), [c^-, c^+), (c^-, c^+)\}$ with $c^-, c^+ \in \mathcal{Q} \cup \{-\infty, +\infty\}$ in the dense case. Intuitively, the semantics of \square_I and \diamond_I are as follows: The formula $\square_I A$ is true if A is true in all moments in the interval I , whereas the formula $\diamond_I A$ is true if A is true in some moment in the interval I . As an example, consider the following simple MTL-program:

```

□[6,44] salary(peter, 55000).
□[45,49] salary(peter, 70000).
    ⋮
□[6,39] department(peter, shoes).
□[40,45] department(peter, clothing).
□[36,46] department(john, toys).
    ⋮
□[6,39] manager(peter, shoes).
□[40,44] manager(paul, shoes).
    ⋮
□ is_manager(X) ← manager_of(X, D).

```

The meaning of the predicates `salary`, `department`, `manager`, and `is_manager` in the above program is as follows: `salary(X,Y)`: the salary of X is Y, `department(X,Y)`: X works in the department Y, `manager(X,Y)`: X is the manager of the department Y, and `is_manager(X)`: X is a manager.

Concerning the operational semantics of simple MTL-programs, a transformation that translates the MTL-programs into classical constraint logic programs has been defined. The translation adds an additional argument representing time to each program predicate. Moreover, the temporal relations which are expressed by temporal operators in MTL-programs are now expressed by constraints in the body of the clauses of the constraint logic program. The proof system formed in this way, called *MTL-resolution* [Brz98], has been proved to be sound and complete.

3.1.5 Clocked Temporal Logic Programming

A temporal extension of logic programming based on a *Clocked Temporal Logic*, a linear-time temporal logic with a multiple granularity of time, is proposed in [LO95, LO96]. The language, called Chronolog(MC), is an extension of Chronolog. In Clocked Temporal Logic, predicates are associated with *local clocks*. Local clocks can be used to model the multiple granularity of time. A Chronolog(MC) program consists of three parts: a *clock definition*, a *clock assignment*, and a *program body*. The clock definition specifies all the local clocks involved in the program, whereas the clock assignment assigns clocks for all the predicate symbols in the program body. Finally, the program body consists of rules and facts.

The following simple example called the Cat-and-Mouse system, showing the expressive power of Chronolog(MC), is adapted from [LO96]. In this example, it is supposed that there is a room where it is quiet only at odd moments in time when nobody occupies it. The room is occupied at even moments in time by a cat and a mouse in turns. In the room, there is also an alarm making sounds. The interval between two sounds is 3 time units. The alarm makes a long sound and a short sound alternately. The first sound made is short. Three predicates are defined in the program. The predicate `occupies(X)`: X occupies the room, the predicate `quiet`: it is quiet in the room, and the predicate `alarm(X)`: X is either `long` or `short`. In the clock definition part of the program, three clocks namely `clock1`, `clock2`, and `clock3`, are defined:

```

first clock1(0).
next clock1(Z) ← clock1(Y), Z is Y + 2.
first clock2(1).
next clock2(Z) ← clock2(M), Z is Y + 2.
first clock3(1).
next clock3(Z) ← clock3(M), Z is Y + 3.

```

In the clock assignment, clocks are assigned to predicates `occupies`, `quiet`, and `alarm`:

```
assign_clock(occupies, clock1).
assign_clock(quiet, clock2).
assign_clock(alarm, clock3).
```

Finally, in the program body, the predicates `occupies`, `quiet`, and `alarm` are defined:

```
first occupies(mouse).
next occupies(cat) ← occupies(mouse).
next occupies(mouse) ← occupies(cat).
first quiet.
next quiet ← quiet.
first alarm(short).
next alarm(long) ← alarm(short).
next alarm(short) ← alarm(long).
```

Declarative and operational semantics have been introduced for Chronolog(MC) in [LO96, LO95]. The *minimal model semantics* of Chronolog(MC) are based on the notion of *clocked temporal Herbrand interpretations*, which extends the notion of temporal Herbrand interpretations of Chronolog to take into account the clocks.

The operational semantics of Chronolog(MC) is based on a resolution-type proof procedure, called *Clocked TiSLD-resolution* [LO96, LO95]. Clocked TiSLD-resolution extends TiSLD-resolution of Chronolog in that it introduces a time matching in the unification of two temporal atoms which ensures that the current time in the matching atoms, which may have been assigned different clocks, is the same.

Applications of Chronolog(MC) are proposed in [LO97a, LO97b]. More specifically, in [LO97a] Chronolog(MC) is used as a specification language to describe concurrent systems, whereas in [LO97b] Chronolog(MC) is used as a language for the specification of distributed computations.

3.2 Branching-Time Temporal Logic Programming

Most of the temporal logic programming languages proposed in the literature are based on linear-time temporal logics. However, there are also temporal logic programming languages based on branching-time logics.

3.2.1 The Language Cactus

A temporal logic programming language called Cactus is proposed in [RGP97, RGP98, GRP97a]. Cactus is based on a discrete, treelike notion of time; that is, every moment in

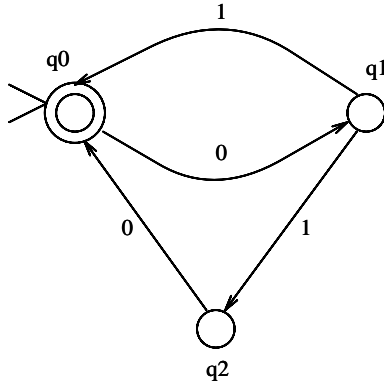


Figure 1: A nondeterministic finite automaton

time may have more than one immediate next moments. Cactus uses the set $List(\mathcal{N})$ of lists of natural numbers \mathcal{N} as the collection of moments in time.

Cactus supports two main operators: the temporal operator **first**, which refers to the beginning of time (or, alternatively, to the root of the tree), and the temporal operator **next_i**, which refers to the i -th child of the current moment (or, alternatively, the i -th branch of the current node in the tree). Note that we actually have a family $\{\text{next}_i \mid i \in \mathcal{N}\}$ of **next** operators, each one of them representing a different next moment that immediately follow the present one.

A *Cactus program* is a set of temporal Horn clauses (i.e., Horn clauses whose atoms may be preceded by one or more temporal operators).

As an example, consider the following Cactus program which describes the behavior of the nondeterministic finite automaton shown in Figure 1. The automaton accepts the regular language $L = (01 \cup 010)^*$:

```

first state(q0).
next0 state(q1) ← state(q0).
next1 state(q0) ← state(q1).
next1 state(q2) ← state(q1).
next0 state(q0) ← state(q2).

```

Note that **q0** is both the initial and the final state. Posing the goal clause,

```

← first next0 next1 next0 state(q0).

```

to the above program will return the answer **yes**, which indicates that the string 010 belongs to the language L . Cactus is appropriate for describing nondeterministic computations or, more generally, computations that involve the manipulation of trees [RGP97, RGP98]. Minimal temporal model and fixpoint semantics have been defined for Cactus programs [RGP98]. As in Chronolog, the declarative semantics of Cactus are based

on the notion of *temporal Herbrand interpretations*. The minimal temporal Herbrand model of a Cactus program contains all ground canonical temporal atoms that are logical consequences of the program.

A refutation proof procedure for Cactus programs, called *BSLD-resolution*, is presented in [RGP97]. BSLB-resolution is similar to TiSLD-resolution [OW93] developed for Chronolog programs and is applied to canonical program and goal clauses. A more general proof procedure which, unlike BSLD-resolution, does not require the program and goal clauses to be in their canonical form is presented in [GRP97a]. Finally, extensions of Cactus have been studied in [Ger99].

4 Modal Logic Programming

A considerable amount of research work has been done in the area of the definition of logic programming languages based on modal logics. Modal logic programming languages have been proven useful in a wide variety of applications, including module definition, agent programming, and representation of actions. In this section, we present two representative instances of modal logic programming languages.

4.1 A Modal Language and Its Use for Module Definition

In [BGM94, BGM98] a modal extension of Horn clause logic is defined which allows both multiple modalities and embedded implications. The syntax of the language is given by the following BNF formulas:

$$\begin{aligned} G &::= T \mid A \mid G_1 \wedge G_2 \mid \exists x G \mid [a_i]G \mid \Box G \mid [a_i](G \leftarrow D) \mid \Box(G \leftarrow D) \\ D &::= H \leftarrow G \mid D_1 \wedge D_2 \mid [a_i]D \mid \Box D \mid \forall x D \\ H &::= A \mid [a_i]H \mid \Box H \end{aligned}$$

where A stands for an atomic formula, T for the proposition *true*, G for a goal, D for a clause, and H for a clause head. A program consists of a set of clauses D . From the above BNF formulas, it is easy to see that sequences of modal operators $[a_i]$ and \Box can freely occur in front of clauses, in front of clause heads, in front of each goal, and in front of embedded implications. The modal operators $[a_1], \dots, [a_k]$ are modalities of type K , whereas the operator \Box is a modality of type $S4$. The operator \Box , which is intended to represent what is known by all agents, interacts with each operator $[a_i]$ through the following *interaction axioms*:

$$\Box A \rightarrow [a_i]A$$

Notice that [Che80, Gol92], for a modality $[t]$ of type K the axiom schema K is assumed:

$$K(t) : [t](a \rightarrow b) \rightarrow ([t]a \rightarrow [t]b)$$

whereas for a modality $[t]$ of type $S4$ the axiom schemas K , T , and 4 are assumed:

$$\begin{aligned} T(t) : [t]a &\rightarrow a \\ 4(t) : [t]a &\rightarrow [t][t]a \end{aligned}$$

A sound and complete goal-directed proof procedure for the proposed language is defined in [BGM98]. Alternatively, in [BGM96b] a translation method is presented from programs in the above modal logic programming language into classical Horn clause programs.

The constructs provided by the proposed modal logic programming language are suitable for structuring knowledge. An interesting application of the language is its ability to define modules through the use of multiple modalities [BGM98]. This can be done by associating a modality $[a_i]$ with each module. Then, $[a_i]$ can be regarded as a module name and can be used to represent what is true in the module. This provides a simple way to define a flat collection of modules and to specify the proof of a goal in a module. In particular, if D is a set of clauses, then the *module definition*

$$\Box[a_i]D$$

defines the clauses in D as belonging to module a_i . The modality \Box in front of the module definition is needed to make the definition visible in any context (i.e., from inside other modules). To prove a goal G in a module a_i , we have to write the goal $[a_i]G$. The clauses in a module must have the form $A \leftarrow G$, where G may contain occurrences of goals $[a_i]G$.

As an example, consider the following program which collects the tree nodes in a list (preorder traversal) defined as a two module program:

```

 $\Box[\text{tree}]\{$ 
    preorder(t(X, LT, RT), [X|L])  $\leftarrow$  preorder(LT, LL),
    preorder(RT, RL),
    [list]append(LL, RL, L).
    preorder(void, []).}
 $\Box[\text{list}]\{$ 
    append([], X, X).
    append([X|Y], Z, [X|W])  $\leftarrow$  append(Y, Z, W).
    ...}

```

The module $[\text{list}]$ contains the definition of `append`, whereas the module $[\text{tree}]$ contains the definition of the predicate `preorder`.

The goal $\leftarrow [\text{tree}]\text{preorder}(t(3, t(4, \text{void}, t(5, \text{void}, \text{void})), t(6, \text{void}, \text{void})), L)$ succeeds with the answer $L = [3, 4, 5, 6]$.

Although, in the above view modules are closed environments and they cannot be composed, the language proposed in [BGM98] also enables one to define modules as

open environments. An open module is allowed to *export* information to the external environment. The modal operator \Box controls the information that is exported by a module. Consider the following three types of clause allowed by the language and their use in module definition. Clauses of the form

$$A \leftarrow G$$

are **local** to the module in which they are defined. Clauses of the form

$$\Box A \leftarrow G$$

export their head and they are called *static*; clauses of the form

$$\Box (A \leftarrow G)$$

are wholly exported by the module and they are called *dynamic*. Using the syntax offered by the language, one can obtain different forms of module composition. The definition of nested modules is also allowed.

The language can also be used to reason in a multiagent situation, as a modality $[a_i]$, regarded as *belief* operator, can be associated with each agent. The operator \Box can be regarded as a kind of *common knowledge* operator.

4.2 A Framework for Developing Modal Logic Programming Languages

In [BGM96a], a framework for developing modal extensions of logic programming is presented. The framework is based on a normal modal logic [Che80] and is parametric with respect to the properties chosen for the modalities and allows sequences of modalities of the form $[t]$, where t is a term of the language, to occur in front of goals, clauses, and clause heads. Because the logic is normal, the axiom schema K is assumed for all modalities $[t]$. The framework allows the use of modalities whose properties can be characterized by axioms of the form

$$[t_1] \dots [t_n]a \rightarrow [s_1] \dots [s_m]a$$

called *inclusion axioms*. For example, consider the modalities $[m]$ and $[p]$ with the meaning “*Mary knows*” and “*Paul knows*”, respectively. Then, the axiom schema

$$[m]a \rightarrow [p]a$$

encodes the statement “*if Mary knows something, then Paul knows the same thing*”. However, the axiom schema

$$[m]a \rightarrow [m][p]a$$

encodes the statement “*if Mary knows something, then she knows that Paul also knows the same thing*”. In addition to the axiom schemas referring to properties of modalities, the programmer writes the clauses of his program. For example, the clauses

$$\begin{aligned} & [m]\text{rains.} \\ & [m](\text{rains} \rightarrow \text{cloudy}) \end{aligned}$$

encode the statements “*Mary knows that it rains*” and “*Mary knows that if it rains, then the weather is cloudy*”.

A sound and complete goal-directed proof procedure, which is modular with respect to the chosen set of axioms, is also presented in [BGM96a]. The proof procedure uses the notion of *matching relation* between sequences of modalities (called *modal context*), which depends only on the properties of modalities themselves.

For example, by the goal clause

$$\leftarrow [p]\text{cloudy}$$

we encode the query “*Does Paul know that it is cloudy?*” This goal clause succeeds.

As indicated in [BGM96a], the proposed framework can be used to reason about actions, to define parametric and nested modules as well as to express inheritance and hierarchy.

5 Other Temporal and Modal Logic Programming Languages

In the previous sections, we have presented representative temporal and modal logic programming languages. It is, however, difficult to present all languages of this family in an article because the number of them is quite high. In this section, we give references to some other languages, encouraging the interested reader to consult them.

Concerning temporal logic programming, an extension of Prolog, called *Temporal Prolog*, is proposed by Hrycej [Hry93]. Temporal Prolog is capable of handling temporally referenced logical statements and temporal constraints. Hrycej bases his work on a fragment of Allen’s interval temporal logic [All83, All84] with a Horn logical axiomatization, which is used as basis for the operational semantics of Temporal Prolog.

Another approach to temporal logic programming proposes [Fru94, Fru96, RF99] the use of *annotated constraint logic programming* for temporal reasoning. In the same direction, we can also see the work presented in [PG95a, PG95b, Pan99].

A framework, called METATEM, for the use of temporal logic as an executable imperative language is presented in [BFG⁺95]. The logic used as a basis of METATEM is a discrete, linear temporal logic. METATEM is based on the idea that future statements can

be seen as commands (*imperative reading*). A concurrent extension of METATEM, called Concurrent METATEM, has been also proposed [Fis94]. Applications of METATEM in modeling reactive systems are investigated in [FFO93]. Applications and extensions of Concurrent METATEM are investigated in [KF97, FW94].

Multidimensional extensions of logic programming languages have been investigated in [OD94, OD97, GR98]. In a multidimensional logic, the values of elements vary depending on more than one dimension, such as time and space. In the multidimensional logic on which the language proposed in [OD94, OD97] is based, for each dimension $k \geq 0$ there are three contextual operators: init_k , prior_k , and rest_k . Informally, init_k refers to the origin along the dimension k , prior_k refers to the previous point along the dimension k , and rest_k refers to the next point along the dimension k . Each dimension is discrete, linear with an initial point. An extension of multidimensional logic programming in which each dimension may be either linear or branching is outlined in [GR98].

In [DEL92] a language called PATHLOG is defined. PATHLOG is based on the idea of translating multimodal logics into classical first-order logic so that standard theorem provers can be used without the need to build new ones. The translation method is based on the idea of making explicit reference to the worlds by adding an extra argument representing the world where the predicate holds.

Temporal logic languages suitable for (deductive) database applications have been investigated in [CI88, BCW93, Cho94, Org96, OM93]. Chomicki and Imielinski [CI88] considered a temporal extension of Datalog (logic programs without function symbols), called Datalog_{1S}, obtained by tagging each predicate with an additional argument modeling time. In this argument, which is a compound term, the successor function symbol is used to construct terms modelling time points. A generalization of Datalog_{1S}, called Datalog_{nS}, is presented in [Cho94] and the bottom-up evaluation of Datalog_{nS} programs is investigated. In [Org96, OM93], a language suitable for deductive database applications, called Temporal Datalog, is investigated. A program in Temporal Datalog is a Chronolog program without function symbols.

Other approaches to temporal and modal logic programming can be found in [Hal87, Gab87]. MOLOG is another modal logic programming language proposed in the literature [dC86]. Finally, for surveys on temporal and modal logic programming languages, the interested reader may refer to [OM94, FO95, Org94].

6 Conclusions

The basic ideas of modal and temporal logic programming languages have been presented in this article. Representative programming languages are briefly presented and simple examples of their use are given.

The history of the field of temporal and modal logic programming started about 15

years ago, and the number of the programming languages proposed in the literature is quite high. However, up until now, none of these languages seems to have the success of Prolog, the first logic programming language. As an explanation, we can refer to the fact that the efficiency of the proposed systems is still far from being acceptable for a language used for large-scale applications. Moreover, most of these languages have been proposed with a specific range of applications in mind; thus, these languages are far from being real general-purpose languages.

On the other hand, temporal and modal logic programming languages have (in most cases) a clear, declarative nature and mathematically defined semantics. The research area of temporal and modal logic programming is an active and evolving research area, with considerable influence on the design of programming languages.

Acknowledgements

I would like to thank P. Rondogiannis, T. Mitakos, and C. Nomikos for their helpful comments.

References

- [All83] J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of ACM*, 26(11):832–843, 1983.
- [All84] J. F. Allen. Towards a general theory of action and time. *Artificial Intelligence*, 23:123–154, 1984.
- [AM89] M. Abadi and Z. Manna. Temporal logic programming. *Journal of Symbolic Computation*, 8:277–295, 1989.
- [Bau93] M. Baudinet. A simple proof of the completeness of temporal logic programming. In L. Farinas del Cerro and M. Penttonen, editors, *Intensional Logics for Programming*, pages 51–83. Oxford University Press, 1993.
- [BCW93] M. Baudinet, J. Chomicki, and P. Wolper. Temporal deductive databases. In L. Farinas del Cerro and M. Penttonen, editors, *Temporal Databases: Theory, Design, and Implementation*, pages 294–320. The Benjamin/Cummings Publishing Company, Inc, 1993.
- [BFG⁺95] H. Barringer, M. Fisher, D. M. Gabbay, G. Gough, and R. Owens. MetateM: An imperative approach to temporal logic programming. *Formal Aspects of Computing*, 7(5):111–154, 1995.

- [BGM94] M. Baldoni, L. Giordano, and A. Martelli. A modal extension of logic programming. In M. Alpuente, R. Barbuni, and I. Ramos, editors, *Proc. of the 1994 Joint Conference on Declarative Programming (GULP-PROBE)*, pages 324–335, 1994.
- [BGM96a] M. Baldoni, L. Giordano, and A. Martelli. A framework for modal logic programming. In Michael Maher, editor, *Proc. of the 1996 Joint International Conference and Symposium on Logic Programming*, pages 52–66. MIT Press, 1996.
- [BGM96b] M. Baldoni, L. Giordano, and A. Martelli. Translating a modal language with embedded implication into horn clause logic. In *Proc. of the International Workshop on Extensions of Logic Programming (ELP'96)*, Lecture Notes in Artificial Intelligence, Vol. 1050, pages 19–33, 1996.
- [BGM98] M. Baldoni, L. Giordano, and A. Martelli. A modal extension of logic programming: Modularity, beliefs and hypothetical reasoning. *Journal of Logic and Computation*, 8(5):597–635, 1998.
- [Brz91] C. Brzoska. Temporal logic programming and its relation to constraint logic programming. In V. Saraswat and K. Ueda, editors, *Logic Programming: Proc. of the 1991 International Symposium*, pages 661–677. MIT Press, 1991.
- [Brz93] C. Brzoska. Temporal logic programming with bounded universal modality goals. In D. S. Warren, editor, *Proc. of the Tenth International Conference on Logic Programming*, pages 239–256. MIT Press, 1993.
- [Brz98] Christoph Brzoska. Programming in metric temporal logic. *Theoretical Computer Science*, 202(1-2):55–125, 1998.
- [Che80] Brian F. Chellas. *Modal Logic: An Introduction*. Cambridge University Press, 1980.
- [Cho94] J. Chomicki. Temporal query languages: a survey. In D. M. Gabbay and H. J. Ohlbach, editors, *Proc. of the First International Conference on Temporal Logic*, Lecture Notes in Artificial Intelligence (LNAI), Vol 827, pages 506–534. Springer-Verlag, 1994.
- [CI88] J. Chomicki and T. Imielinski. Temporal deductive databases and infinite objects. In *Proceedings of the Seventh ACM SIGART-SIGMOND-SIGART Symposium on Principles of Database Systems*, pages 61–73. ACM Press, 1988.
- [dC86] Luis Farinas del Cerro. MOLOG: A system that extends PROLOG with modal logic. *New Generation Computing*, 4:35–50, 1986.

- [DEL92] F. Debart, P. Enjalbert, and M. Lescot. Multimodal logic programming using equational and order-sorted logic. *Theoretical Computer Science*, 105(1):141–166, 1992.
- [FFO93] M. Finger, M. Fisher, and R. Owens. METATEM at work: Modelling Reactive Systems using Executable Temporal Logic. In *Proc. of the Sixth International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert-Systems (IEA-AIE), Edinburgh, UK, June, 1993*.
- [Fis94] Michael Fisher. A survey of concurrent METATEM-The language and its applications. In D. M. Gabbay and H. J. Ohlbach, editors, *Proc. of the First International Conference on Temporal Logics (ICTL'94)*, Lecture Notes in Artificial Intelligence (LNAI), Vol 827, pages 480–505. Springer-Verlag, 1994.
- [FO95] M. Fisher and R. Owens. An introduction to executable modal and temporal logics. Lecture Notes in Artificial Intelligence (LNAI) 897. Springer-Verlag, February 1995.
- [Fru94] T. Fruehwirth. Annotated constraint logic programming applied to temporal reasoning. In *Proc. of the International Symposium on Programming Language Implementation and Logic Programming (PLILP'94)*, Lecture Notes in Computer Science (LNCS) 844, pages 230–243. Springer-Verlag, 1994.
- [Fru96] Thom Fruehwirth. Temporal annotated constraint logic programming. *Journal of Symbolic Computation*, 22:555–583, 1996.
- [FW94] M. Fisher and M. Wooldridge. Specifying and executing protocols for cooperative action. In *International Working Conference on Cooperating Knowledge-Based Systems (CKBS), Keele, UK, June 1994*.
- [Gab87] Dov Gabbay. Modal and temporal logic programming. In A. Galton, editor, *Temporal Logics and their applications*, pages 197–237. Academic Press, London, 1987.
- [Ger99] Manolis Gergatsoulis. Extending the branching-time logic programming language Cactus. In M. Gergatsoulis and P. Rondogiannis, editors, *Proc. of the 12th International Symposium on Languages for Intensional Programming (ISLIP'99)*, pages 232–245, 1999.
- [GHR94] D. M. Gabbay, I. Hodkinson, and M. Reynolds. *Temporal Logic: Mathematical foundations and computational aspects*. Clarendon Press-Oxford, 1994.
- [Gol92] Robert Goldblatt. *Logics of Time and Computation (2nd edition)*. CSLI Lecture Notes Nr. 7, 1992.

- [GR98] M. Gergatsoulis and P. Rondogiannis. Temporal and multidimensional logic programming languages. In *Proc. of the 2nd Conference on Technology and Automation, October 2-3, Thessaloniki, Greece*, pages 223–227, 1998.
- [GRP96] M. Gergatsoulis, P. Rondogiannis, and T. Panayiotopoulos. Disjunctive Chronolog. In M. Chacravarty, Y. Guo, and T. Ida, editors, *Proceedings of the JICSLP'96 Post-Conference Workshop "Multi-Paradigm Logic Programming"*, pages 129–136, Bonn, 5-6 Sept. 1996.
- [GRP97a] M. Gergatsoulis, P. Rondogiannis, and T. Panayiotopoulos. Proof procedures for branching-time logic programs. In W. W. Wadge, editor, *Proc. of the Tenth International Symposium on Languages for Intensional Programming (ISLIP'97), May 15-17, Victoria BC, Canada*, pages 12–26, 1997.
- [GRP97b] M. Gergatsoulis, P. Rondogiannis, and T. Panayiotopoulos. Temporal disjunctive logic programming. Technical Report 15-97, Dept. of Computer Science, University of Ioannina, Greece, 1997.
- [Hal87] Roger Hale. Temporal logic programming. In A. Galton, editor, *Temporal Logics and their applications*, pages 91–119. Academic Press, London, 1987.
- [Hry93] T. Hrycej. A temporal extension of Prolog. *The Journal of Logic Programming*, 15:113–145, 1993.
- [JM94] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *The Journal of Logic Programming*, 19/20:503–582, May/July 1994.
- [KF97] A. Kellet and M. Fisher. Concurrent METATEM as a Coordination Language. In *Coordination Languages and Models*, Lecture Notes in Computer Science (LNCS), Vol. 1282. Springer Verlag, 1997.
- [Llo87] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.
- [LMR92] J. Lobo, J. Minker, and A. Rajasekar. *Foundations of Disjunctive Logic Programming*. MIT Press, 1992.
- [LO95] C. Liu and M. A. Orgun. Dealing with multiple granularity of time in temporal logic programming. Technical Report TR95-04, Dept. of Computing, Macquarie University, NSW 2109, Australia, October 1995.
- [LO96] C. Liu and M. A. Orgun. Clocked temporal logic programming. In *Proc. of the 19th Australasian Computer Science Conference*, pages 272–280, 1996.

- [LO97a] C. Liu and M. Orgun. The specification and verification of concurrent systems with Chronolog(MC). In W. W. Wadge, editor, *Proc. of the Tenth International Symposium on Languages for Intensional Programming (ISLIP'97), May 15-17, Victoria BC, Canada*, pages 27–33, 1997.
- [LO97b] C. Liu and M. A. Orgun. Structural simulation of distributed computations using Chronolog(MC). Technical Report C/TR97-07, Dept. of Computing, Macquarie University, NSW 2109, Australia, May 1997.
- [LR91] J. Lobo and A. Rajasekar. Semantics of Horn and disjunctive logic programs. *Theoretical Computer Science*, 86(1):93–106, 1991.
- [MRL91] J. Minker, A. Rajasekar, and J. Lobo. Theory of disjunctive logic programs. In J. L. Lasser and G. Plotkin, editors, *Computational Logic. Essays in the Honor of Alan Robinson*, pages 613–639. MIT Press, 1991.
- [NM94] G. Nadathur and D. Miller. Higher-order logic programming. Technical Report CS-1994-38, Dept. of Computer Science, Duke University, December 1994.
- [OD94] M. A. Orgun and W. Du. Multi-dimensional logic programming. *Journal of Computing and Information*, 1(1):1501–1520, 1994. Special Issue: Proc. of the 6th International Conf. on Computing and Information.
- [OD97] M. A. Orgun and W. Du. Multi-dimensional logic programming: Theoretical foundations. *Theoretical Computer Science*, 158(2):319–345, 1997.
- [OM93] M. A. Orgun and H. A. Muller. A temporal algebra based on an abstract model. In *Advances in Database Research: Proc. of the 4th Australian Database Conference (Brisbane, Queensland, February 1-2)*, pages 301–316. World Scientific, Singapore, 1993.
- [OM94] M. A. Orgun and W. Ma. An overview of temporal and modal logic programming. In D. M. Gabbay and H. J. Ohlbach, editors, *Proc. of the First International Conference on Temporal Logics (ICTL'94)*, Lecture Notes in Artificial Intelligence (LNAI), Vol 827, pages 445–479. Springer-Verlag, 1994.
- [Org91] M. A. Orgun. *Intensional logic programming*. PhD thesis, Dept. of Computer Science, University of Victoria, Canada, December 1991.
- [Org94] M. A. Orgun. Temporal and modal logic programming: An annotated bibliography. *SIGART Bulletin*, 5(3):52–59, 1994.
- [Org96] M. A. Orgun. On temporal deductive databases. *Computational Intelligence*, 12(2):235–259, 1996.

- [OW92a] M. A. Orgun and W. W. Wadge. Theory and practice of temporal logic programming. In L. Farinas del Cerro and M. Penttonen, editors, *Intensional Logics for Programming*, pages 23–50. Oxford University Press, 1992.
- [OW92b] M. A. Orgun and W. W. Wadge. Towards a unified theory of intensional logic programming. *The Journal of Logic Programming*, 13(4):413–440, August 1992.
- [OW93] M. A. Orgun and W. W. Wadge. Chronolog admits a complete proof procedure. In *Proc. of the Sixth International Symposium on Logic and Intensional Programming (ISLIP'93)*, pages 120–135, 1993.
- [OW94] M. A. Orgun and W. W. Wadge. Extending temporal logic programming with choice predicates non-determinism. *Journal Logic and Computation*, 4(6):877–903, 1994.
- [OWD93] M. A. Orgun, W. W. Wadge, and W. Du. Chronolog(\mathcal{Z}): Linear-time logic programming. In O. Abou-Rabia, C. K. Chang, and W. W. Koczkodaj, editors, *Proc. of the Fifth International Conference on Computing and Information*, pages 545–549. IEEE Computer Society Press, 1993.
- [Pan99] Themis Panayiotopoulos. TRL: A temporal reasoning system and its applications. In M. Gergatsoulis and P. Rondogiannis, editors, *Proc. of the 12th International Symposium on Languages for Intensional Programming (ISLIP'99)*, pages 246–261, 1999.
- [PG95a] T. Panayiotopoulos and M. Gergatsoulis. Intelligent information processing using TRLi. In A. Min Tjoa N. Revell, editor, *6th International Conference and Workshop on Data Base and Expert Systems Applications (DEXA' 95), (Workshop Proceedings), London, UK, 4-8 September*, pages 494–501, 1995.
- [PG95b] T. Panayiotopoulos and M. Gergatsoulis. A Prolog like temporal reasoning system. In M. H. Hamza, editor, *Proc. of 13th IASTED International Conference on APPLIED INFORMATICS, ICLS(Innsbruck), Austria 21-23 February*, pages 123–126, 1995.
- [RF99] A. Raffaeta and T. Fruehwirth. Two semantics for temporal annotated constraint logic programming. In M. Gergatsoulis and P. Rondogiannis, editors, *Proc. of the 12th International Symposium on Languages for Intensional Programming (ISLIP'99)*, pages 126–140, 1999.

- [RGP97] P. Rondogiannis, M. Gergatsoulis, and T. Panayiotopoulos. *Cactus: A branching-time logic programming language*. In D. Gabbay, R. Kruse, A. Nonnengart, and H. J. Ohlbach, editors, *Proc. of the First International Joint Conference on Qualitative and Quantitative Practical Reasoning, ECSQARU-FAPR'97, Bad Honnef, Germany*, Lecture Notes in Artificial Intelligence (LNAI) 1244, pages 511–524. Springer, June 1997.
- [RGP98] P. Rondogiannis, M. Gergatsoulis, and T. Panayiotopoulos. Branching-time logic programming: The language Cactus and its applications. *Computer Languages*, 24(3):155–178, October 1998.
- [vEK76] M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *J. ACM*, 23(4):733–742, Oct. 1976.